THE EXPERT'S VOICE® IN WEB DEVELOPMENT

SECOND EDITION

Pro JavaScript Techniques

REAL-WORLD JAVASCRIPT TECHNIQUES FOR TODAY'S DEVELOPER

John Resig, Russ Ferguson, and John Paxton

Apress®

THE EXPERT'S VOICE® IN WEB DEVELOPMENT

SECOND EDITION

Pro JavaScript Techniques

REAL-WORLD JAVASCRIPT TECHNIQUES FOR TODAY'S DEVELOPER

John Resig, Russ Ferguson, and John Paxton

Apress®

Contents at a Glance

CHAPTER 1

Professional JavaScript Techniques

Welcome to *Pro JavaScript Techniques*. This book provides an overview of the current state of JavaScript, particularly as it applies to the professional programmer. Who is the professional programmer? Someone who has a firm grasp of the basics of JavaScript (and probably several other languages). You are interested in the breadth and depth of JavaScript. You want to look at the typical features like the Document Object Model (DOM), but also learn about what's going on with all this talk of Model-View-Controller (MVC) on the client side. Updated APIs, new features and functionality, and creative applications of code are what you are looking for here.

This is the second edition of this book. Much has changed since the first edition came out in 2006. At that time, JavaScript was going through a somewhat painful transition from being a toy scripting language to being a language that was useful and effective for several different tasks. It was, if you will, JavaScript's adolescence. Now, JavaScript is at the end of another transition: to continue the metaphor, from adolescence to adulthood. JavaScript usage is nearly ubiquitous, with anywhere from 85 to 95 percent of websites, depending on whose statistics you believe, having some JavaScript on their main page. Many people speak of JavaScript as the most popular programming language in the world (in the number of people who use it on a regular basis). But more important than mere usage are effectiveness and capability.

JavaScript has transitioned from a toy language (image rollovers! status bar text manipulations!) to an effective, if limited tool (think of client-side form validation), to its current position as a broad-featured programming language no longer limited to mere browsers. Programmers are writing JavaScript tools that provide MVC functionality, which was long the domain of the server, as well as complex data visualizations, template libraries, and more. The list goes on and on. Where in the past, designers would have relied on a .NET or Java Swing client to provide a full-featured, rich interface to server-side data, we can now realize that application in JavaScript with a browser. And, using Node.js, we have JavaScript's own version of a virtual machine, an executable that can run any number of different applications, all written in JavaScript and none requiring a browser.

This chapter will describe how we got here and where we are going. It will look at the various improvements in browser technology (and popularity) that have abetted the JavaScript Revolution. The state of JavaScript itself needs inspection, as we want to know where we are before we look at where we are going. Then, as we examine the chapters to come, you will see what the professional JavaScript programmer needs to know to live up to his or her title.

How Did We Get Here?

As of the first edition of the book, Google Chrome and Mozilla Firefox were relatively new kids on the block. Internet Explorer 6 and 7 ruled the roost, with version 8 gaining some popularity. Several factors combined to jump-start JavaScript development.

For most of its life, JavaScript was dependent upon the browser. The browser is the runtime environment for JavaScript, and a programmer's access to JavaScript functionality was highly dependent upon the make, model, and version of browser visiting said programmer's website. By the mid-2000s, the browser wars of the 90s had been easily won by Internet Explorer, and browser development stagnated. Two browsers challenged this state of affairs: Mozilla Firefox and Google Chrome. Firefox was the descendant of Netscape, one of the earliest web browsers. Chrome had Google's backing, more than enough to make it an instant player on the scene.

But both of these browsers made a few design decisions that facilitated the JavaScript revolution. The first decision was to support the World Wide Web consortium's implementation of various standards. Whether dealing with the DOM, event handling, or Ajax, Chrome and Firefox generally followed the spec and implemented it as well as possible. For programmers, this meant that we didn't have to write separate code for Firefox and Chrome. We were already used to writing separate code for IE and something else, so having branching code in itself was not new. But making sure that the branching was not overly complex was a welcome relief.

Speaking of standards, Firefox and Chrome also put in a lot of work with the European Computer Manufacturer's Association (ECMA, now styled Ecma). Ecma is the standards body that oversees JavaScript. (To be technical, Ecma oversees the ECMAScript standard, since JavaScript is a trademark of Oracle and… well, we don't really care about those details, do we? We will use JavaScript to refer to the language and ECMAScript to refer to the specification to which a JavaScript implementation adheres.) ECMAScript standards had languished in much the same way as IE development. With the rise of real browser competition, the ECMAScript standard was taken up again. ECMAScript version 5 (2009) codified many of the changes that had been made in the ten years (!) since the previous version of the standard. The group itself was also energized, with version 5.1 coming out in 2011. The future is provided for, with significant work currently being done on both versions 6 and 7 of the standard.

To give credit where credit is due, Chrome pushed the updating of JavaScript as well. The Chrome JavaScript engine, called V8, was a very important part of Chrome's debut in 2008. The Chrome team built an engine that was much faster than most JavaScript engines, and it has kept that goal at the top of the list for subsequent versions. In fact, the V8 engine was so impressive that it became the core of Node.js, a browser-independent JavaScript interpreter. Originally intended as a server that would use JavaScript as its main application language, Node has become a flexible platform for running any number of JavaScript-based applications.

Back to Chrome: the other major innovation Google introduced to the land of browsers was the concept of the evergreen application. Instead of having to download a separate browser install for updates, Chrome's default is to automatically update the browser for you. While this approach is sometimes a pain in the corporate world, it is a great boon to the noncorporate consumer surfer (also known as a person!). If you use Chrome (and, for the last few years, Firefox), your browser is up-to-date, without your having to make any effort. While Microsoft has done this for a long time in pushing security updates via Windows Update, it does not introduce new features to Internet Explorer unless they are coupled to a new version of Windows. To put it another way, updates to IE are slow in coming. Chrome and Firefox always have the latest and greatest features, as well as being quite secure.

As Google pressed on with Chrome's features, the other browser makers played catch-up. Sometimes this came in sillier ways, such as when Firefox adapted Chrome's version numbering. But it also resulted in Mozilla and Microsoft taking a cold, hard look at JavaScript engines. Both browser makers have significantly overhauled their JS engines over the last few years, and Chrome's lead, while formidable, is no longer insurmountable.

Finally, Microsoft has (mostly) thrown in the towel on its classic "embrace and extend" philosophy, at least when it comes to JavaScript. With IE version 9, Microsoft implemented World Wide Web Consortium (W3C) event handling and standardized its DOM interfaces as well as its Ajax API. For most of the standard features of JavaScript, we no longer have to implement two versions of the same code! (Legacy code for legacy browsers is still a bit of an issue, of course…)

It seems almost a panacea. JavaScript is faster than ever before. It is easier to write code for a variety of different browsers. Standards documents both describe the real world and provide a useful roadmap to features to come. And most of our browsers are fully up-to-date. So what do we need to worry about now, and where are we going in the future?

Modern JavaScript

It has never been easier to develop serious applications with JavaScript. We have a clear, clean break with the bad old days of separate code for multiple browsers, poor standards poorly implemented, and slow JavaScript engines that were often an afterthought. Let's take a look at the state of the modern JavaScript environment. Specifically, we will look at two areas: the modern browser and the modern toolkit.

Modern JavaScript depends on the idea of the modern browser. What is the modern browser? Different organizations describe it in different ways. Google says that their applications support the current and previous major versions of browsers. (Fascinating, as Gmail still works on IE9, as far as we can tell!) In an interesting article, the people behind the British Broadcasting Company (BBC) website revealed that they define a modern browser as one that supports the following capabilities:

1. `document.querySelector()` / `document.querySelectorAll()`
2. `window.addEventListener()`
3. The Storage API (`localStorage` and `sessionStorage`)

jQuery, probably the most popular JavaScript library on the web, split its versions into the 1.x line, which supports IE 6 and later, and the 2.x line, which supports "modern" browsers like IE 9 and later. And make no mistake, IE is the dividing line between the modern and the ancient. The other two major browsers are evergreen. And while Safari and Opera are not evergreen, they update on a faster schedule than IE and don't have nearly the market share it does.

So where is the borderline for the modern browser? Alas, the border seems to wander between Internet Explorer versions 9 through 11. But IE 8 is definitely on the far side of browser history. It does not support most of the features of ECMAScript 5. It does not include the API for W3C event handling. The list goes on and on. So when we discuss modern browsers, we will refer to at least Internet Explorer 9. And our coverage will not endeavor to support ancient browsers. Where relevant and simple, we will point out polyfills for older versions of IE, but in general, our floor is Internet Explorer 9.

The Rise of Libraries

In addition to the modern browser, there is another important aspect of the current environment for JavaScript we need to discuss: libraries. Over the past 8 years, there has been an explosion in the number and variety of JavaScript libraries. There are more than 800,000 GitHub repositories for JavaScript; of these, almost 900 have more than 1,000 stars. From its humble beginnings as collections of utility functions, the JavaScript library ecosystem has evolved (somewhat chaotically) into a vast landscape of possibilities.

How does this affect us as JavaScript developers? Well, of course, there is the model of "library as expansion," where a library provides additional functionality. Think of the MVC libraries like Backbone and Angular (which we will be looking at in a later chapter), or the data visualization libraries like d3 or Highcharts. But JavaScript is in an interesting position, as libraries can also provide a level interface to features that are standard on some browsers but not on others.

For a long time, the standard example of a variably implemented feature in JavaScript was event handling. Internet Explorer had its own event-handling API. Other browsers generally followed the W3C's API. Various libraries provided unified implementations for event handling, including the best of both worlds. Some of these libraries stood alone, but the successful ones also normalized functionality for Ajax, the DOM, and a number of other features that were differently implemented across browsers.

The most popular of these libraries has been jQuery. Since its inception, jQuery has been the go-to library for using new JavaScript features without worrying about the browser's support for those features. So instead of using IE's event handling or the W3C's, you could simply use jQuery's `.on()` function, which wrapped around the variance, providing a unified interface. Several other libraries provided similar functionality: Dojo, Ext JS, Prototype, YUI, MooTools, and so on. These toolkit libraries aimed to standardize APIs for developers.

The standardization goes further than providing simple branching code. These libraries often ameliorate buggy implementations. The official API for a function may not change much between versions, but there will be bugs; sometimes those bugs will be fixed, sometimes not, and sometimes the fixes will introduce new bugs. Where libraries could fix or work around these bugs, they did. For example, jQuery 1.11 contains more than a half-dozen fixes for problems with the event-handling API.

Some libraries (jQuery in particular) also provided new or different interpretations of certain capabilities. The jQuery selector function, the core of the library, predates the now-standard `querySelector()` and `querySelectorAll()` functions, and it was a driver for including those functions in JavaScript. Other libraries provide access to functionality despite very different underlying implementations. Later in the book, we will look at Ajax's new Cross Origin Resource Sharing (CORS) protocol, which allows for Ajax requests to servers other than the one that originally served the page. Some libraries have already implemented a version of this that uses CORS but falls back to JSON with padding (JSON-P) where needed.

Because of their utility, some libraries have become part of a professional JavaScript programmer's standard development toolkit. Their features may not be standardized into JavaScript (yet), but they are an accumulation of knowledge and functionality that simply makes it easier to realize designs quickly. In recent years, though, you could get quite a few hits to your blog by asking whether jQuery (or another library) was really necessary for development on a modern browser. Consider the BBC's requirements; you can certainly realize a large degree of jQuery-like functionality if you have those three methods available to you. But jQuery also includes a simplified yet expanded DOM interface, it handles bugs for a variety of different edge cases, and if you need support for IE 8 or earlier, jQuery is your major option. Accordingly, the professional JavaScript programmer must look at the requirements for a project and consider whether it pays to risk reinventing the wheel that jQuery (or another similar library) provides.

More Than a Note about Mobile

In older JavaScript and web development books, you would reliably see a section, maybe a whole *chapter*, on what to do about mobile browsing. Mobile browsing was a small enough share of total browsing, and the market was so fractured, that it seemed only specialists would be interested in mobile development. That's not the case anymore. Since the first edition of this book, mobile web browsing has exploded, and it is a very different beast from desktop development. Consider some statistics: according to a variety of sources, mobile browsing represents between 20 and 30 percent of all browsing. By the time you are reading this, it may well represent more, as it has consistently increased since the debut of the iPhone. Speaking of which, well over 40 percent of mobile browsing is done with iOS Safari, although Android's browser and Chrome for Android are gaining ground (and may have overtaken Safari, depending on whose stats you believe). Safari on iOS is not the same as Safari on the desktop, and the same goes for Android Chrome vs. desktop Chrome and mobile Firefox vs. desktop Firefox. Mobile is mainstream.

The browsers on mobile devices provide a new set of challenges and opportunities. Mobile devices are often more limited than desktops (though that's another gap that is rapidly closing). Conversely, mobile devices offer new features (swipe events, more accurate geolocation, and so on) and new interaction idioms (using the hand instead of the mouse, swiping for scrolling). Depending on your development requirements, you may have to build an app that looks good on mobile as well as the desktop, or reimplement existing functionality for a mobile platform.

The JavaScript landscape has changed extensively over the last few years. Despite some standardization of APIs, there are also many new challenges. How will this affect us as professional JavaScript programmers?

Where Do We Go from Here?

We should set down some standards for ourselves. We have already set one: IE9 as the floor of the modern browser experience. The other browsers are evergreen, and not to worry about. What about mobile, then? While the issue is complex, iOS 6 and Android 4.1 (Jelly Bean) will, in general, serve as our floors. Mobile computing updates faster and more frequently than desktops do, so we are confident in using these more recent versions of mobile operating systems.

That said, let us digress for a moment to discuss not browser versions, operating systems, or platforms, but your audience. While we can quote statistics all day long, the valuable statistics tell you about *your* audience, not the audience in general. Perhaps you are designing for your employer, who has standardized on IE 10. Or maybe your idea for an application depends heavily on features that only Chrome provides. Or maybe there isn't even a desktop version, but you're aiming for a roll-out to iPads and Android tablets. Consider your target audience. This book is written to be broadly applicable, and your application may be as well. But it would be folly to spend time worrying about bugs in IE9 for that previously mentioned tablet-only application, wouldn't it? Now, back to our standards.

For screenshots and testing, this book will generally prefer Google Chrome. Occasionally, we will demonstrate code on Firefox or Internet Explorer where it is relevant. Chrome, for developers, is the gold standard—not necessarily in user-friendliness, but certainly in the information exposed to the programmer. In a later chapter, we will look at the various developer tools available, scrutinizing not only Chrome, but Firefox (with and without Firebug) and IE as well.

As a standard library, we will refer to jQuery. There are many alternatives, of course, but jQuery wins for two reasons: first, it is the most popular general-use JavaScript library on the web. Second, at least one of the authors (John Resig) has a little bit of history with jQuery, which predisposed the other author (John Paxton) to concede the point of working with it. In updating this book, we have replaced many of the techniques from the previous version with jQuery's library of functionality. In these cases, we are disinclined to reinvent the wheel. As needed, we will refer to the appropriate jQuery functionality. We will, of course, discuss new and exciting techniques, as well!

JavaScript IDEs have updated significantly in the last few years, driven by JavaScript's own rise. The possibilities are too numerous to list here, but there are a few applications of note. John Resig uses a highly customized version of vim for his development environment. John Paxton is a little bit lazier, and has elected to use JetBrains' excellent WebStorm (`http://www.jetbrains.com/webstorm/`) as his IDE. Adobe offers the open source, free Brackets IDE (`http://brackets.io/`), currently at version 1.3. Eclipse is also available, and many people have reported positive results by customizing SublimeText or Emacs to do their bidding. As always, use what you feel most comfortable with.

There are other tools that can assist in JavaScript development. Rather than list them here, we will dedicate a chapter to them later in the book. Which means it's a good time to give an outline of what's to come.

Coming Up Next

Starting with Chapter 2, we will look at the latest and greatest in the JavaScript language. This means looking at new features like those available through the `Object` type, but also reexamining some older concepts like references, functions, scope, and closures. We will lump all of this under the heading of **Features, Functions, and Objects**, but it covers a bit more than that.

Chapter 3 discusses **Creating Reusable Code**. Chapter 2 skips over one of the biggest new features of JavaScript, the `Object.create()` method, and its implications for object-oriented JavaScript code. So in this chapter we will spend time with `Object.create()`, functional constructors, prototypes, and object-oriented concepts as implemented in JavaScript.

Having spent two chapters developing code, we should start thinking about how to manage it. Chapter 4 shows you tools for **Debugging JavaScript Code**. We start by examining browsers and their developer tools.

Chapter 5 begins a sequence discussing some high-usage areas of JavaScript functionality. Here we look at the **Document Object Model**. The DOM API has increased in complexity and has not really become more straightforward since the last edition. But there are new features that we should familiarize ourselves with.

In Chapter 6, we attempt to master Events. The big news here is the standardization of the events API along the lines of the W3C style. This provides us the opportunity to move away from utility libraries and finally go deep into the events API without worrying about large variations between browsers.

One of the first non-toy applications for JavaScript was client-side form validation. Amazingly, it took browser makers over a decade to think about adding functionality to form validation beyond capturing the submit event. When looking in Chapter 7 at **JavaScript and Form Validation**, we will discover that there is a whole new set of functionality for form validation provided by both HTML and JavaScript.

Everyone who develops with JavaScript has spent some time **Introduction to Ajax**. With the introduction of Cross-Origin Resource Sharing (CORS), Ajax functionality has finally moved past the silliest of its restrictions.

Command line tools like Yeoman, Bower, Git and Grunt are covered in **Web Production Tools**. These tools will show us how to quickly add all the files and folders needed. This way we can focus on development.

Chapter 10 covers **AngularJS and Testing**. Using the knowledge gained in the previous chapter, we now start to look at what makes Angular work and how to implement both unit and end to end testing.

Last, Chapter 11 discusses the **Future of JavaScript**. ECMAScript 6 will be settled, more or less, by the time this book goes to press. ECMAScript 7 is in active development. Beyond the basics of where JavaScript is going, we will look at what features you can use right now.

Summary

We spent a lot of this chapter on everything around JavaScript: the platform(s), the history, the IDEs, and so on. We believe that history informs the present. We wanted to explain where we were, and how we got here, to help you understand why JavaScript is where it is, and is what it is, today. Of course, we plan to spend the bulk of this book talking about how JavaScript works, particularly for the professional programmer. We feel quite strongly that this book covers the techniques and APIs that every professional JavaScript programmer should be familiar with. So without further ado…

CHAPTER 2

Features, Functions, and Objects

Objects are the fundamental units of JavaScript. Virtually everything in JavaScript is an object and interacts on an object-oriented level. To build up this solid object-oriented language, JavaScript includes an arsenal of features that make it unique in both its foundation and its capabilities.

This chapter covers some of the most important aspects of the JavaScript language, such as references, scope, closures, and context. These are not necessarily the cornerstones of the language, but the elegant arches, which both support and refine JavaScript. We will delve into the tools available for working with objects as data structures. A dive into the nature of object-oriented JavaScript follows, including a discussion of classes vs. prototypes. Finally, the chapter explores the use of object-oriented JavaScript, including exactly how objects behave and how to create new ones. This is quite possibly the most important chapter in this book if taken to heart, as it will completely change the way you look at JavaScript as a language.

Language Features

JavaScript has a number of features that are fundamental to making the language what it is. There are very few other languages like it. We find the combination of features to fit just right, contributing to a deceptively powerful language.

References and Values

JavaScript variables hold data in one of two ways: by copies and references. Anything that is a primitive value is *copied* into a variable. Primitives are strings, numbers, Booleans, null, and undefined. The most important characteristic of primitives is that they are assigned, copied, and passed to and returned from functions by *value*.

The rest of JavaScript relies on references. Any variable that does not hold one of the aforementioned **primitive** values holds a **reference** to an object. A reference is a pointer to the location in memory of an object (or array, or date, or what-have-you). The actual object (array, date, or whatever) is called the **referent**. This is an incredibly powerful feature, present in many languages. It allows for certain efficiencies: two (or more!) variables do not have their own copies of an object; they simply refer to the same object. Updates to the referent made via one reference are reflected in the other reference. By maintaining sets of references to objects, JavaScript affords you much more flexibility. An example of this is shown in Listing 2-1, where two variables point to the same object, and the modification of the object's contents via one reference is reflected in the other reference.

Listing 2-1. Example of Multiple Variables Referring to a Single Object

```
// Set obj to an empty object
// (Using {} is shorter than 'new Object()')
var obj = {};

// objRef now refers to the other object
var refToObj = obj;

// Modify a property in the original object
obj.oneProperty = true;

// We now see that the change is represented in both variables
// (Since they both refer to the same object)
console.log( obj.oneProperty === refToObj.oneProperty );

// This change goes both ways, since obj and refToObj are both references
refToObj.anotherProperty = 1;
console.log( obj.anotherProperty === refToObj.anotherProperty );
```

Objects have two features: properties and methods. These are often referred to collectively as the *members* of an object. Properties contain the data of an object. Properties can be primitives or objects themselves. Methods are functions that act upon the data of an object. In some discussions of JavaScript, methods are included in the set of properties. But the distinction is often useful.

Self-modifying objects are very rare in JavaScript. Let's look at one popular instance where this occurs. The Array object is able to add additional items to itself using the `push` method. Since, at the core of an Array object, the values are stored as object properties, the result is a situation similar to that shown in Listing 2-1, where an object becomes globally modified (resulting in multiple variables' contents being simultaneously changed). An example of this situation can be found in Listing 2-2.

Listing 2-2. Example of a Self-Modifying Object

```
// Create an array of items
// (Similar to 2-1, using [] is shorter than 'new Array()')
var items = [ 'one', 'two', 'three' ];

// Create a reference to the array of items
var itemsRef = items;

// Add an item to the original array
items.push( 'four' );

// The length of each array should be the same,
// since they both point to the same array object
console.log( items.length == itemsRef.length );
```

It's important to remember that references point only to the referent object, not to another reference. In Perl, for example, it's possible to have a reference point to another variable that also is a reference. In JavaScript, however, it traverses down the reference chain and only points to the core object. An example of this situation can be seen in Listing 2-3, where the physical object is changed but the reference continues to point back to the old object.

Listing 2-3. Changing the Reference of an Object While Maintaining Integrity

```
// Set items to an array (object) of strings
var items = [ 'one', 'two', 'three' ];
// Set itemsRef to a reference to items
var itemsRef = items;

// Set items to equal a new object
items = [ 'new', 'array' ];

// items and itemsRef now point to different objects.
// items points to [ 'new', 'array' ]
// itemsRef points to [ 'one', 'two', 'three' ]
console.log( items !== itemsRef );
```

Finally, let's look at a strange instance that you might think would involve references but does not. When performing string concatenation, the result is always a new string object rather than a modified version of the original string. Because strings (like numbers and Booleans) are primitives, they are not actually referents, and the variables that contain them are not references. This can be seen in Listing 2-4.

Listing 2-4. Example of Object Modification Resulting in a New Object, Not a Self-Modified Object

```
// Set item equal to a new string object
var item = 'test';

// itemRef now refers to the same string object
var itemRef = item;

// Concatenate some new text onto the string object
// NOTE: This creates a new object and does not modify
// the original object.
item += 'ing';

// The values of item and itemRef are NOT equal, as a whole
// new string object has been created
console.log( item != itemRef );
```

Strings are often particularly confusing because they act like objects. You can create instances of strings via a call to `new String`. Strings have properties like `length`. Strings also have methods like `indexOf` and `toUpperCase`. But when interacting with variables or functions, strings are very much primitives.

References can be a tricky subject to wrap your mind around, if you are new to them. Nonetheless, understanding how references work is paramount to writing good, clean JavaScript code. In the next couple of sections we're going to look at features that aren't necessarily new or exciting but are important for writing good, clean code.

Scope

Scope is a tricky feature of JavaScript. Most programming languages have some form of scope; the differences lie in the duration of that scope. There are only two scopes in JavaScript: functional scope and global scope. This is deceptively simple. Functions have their own scope, but blocks (such as `while`, `if`, and `for` statements) do not. This may seem strange if you are coming from a block-scoped language. Listing 2-5 shows an example of the implications of function-scoped code.

Listing 2-5. Example of How the Variable Scope in JavaScript Works

```
// Set a global variable, foo, equal to test
var foo = 'test';

// Within an if block
if ( true ) {
    // Set foo equal to 'new test'
    // NOTE: This still belongs to the global scope!
    var foo = 'new test';
}

// As we can see here, as foo is now equal to 'new test'
console.log( foo === 'new test' );

// Create a function that will modify the variable foo
function test() {
    var foo = 'old test';
}

// However, when called, 'foo' remains within the scope
// of the function
test();

// Which is confirmed, as foo is still equal to 'new test'
console.log( foo === 'new test' );
```

You'll notice that in Listing 2-5, the variables are within the global scope. All globally scoped variables are actually visible as properties of the `window` object in browser-based JavaScript. In other environments, there will be a global context to which all globally-scoped variables belong.

In Listing 2-6 a value is assigned to a variable, `foo`, within the scope of the `test()` function. However, nowhere in Listing 2-6 is the scope of the variable actually declared (using `var foo`). When the `foo` variable isn't explicitly scoped, it will become defined globally, even though it is only intended to be used within the context of the function.

Listing 2-6. Example of Implicit Globally Scoped Variable Declaration

```
// A function in which the value of foo is set
function test() {
    foo = 'test';
}

// Call the function to set the value of foo
test();

// We see that foo is now globally scoped
console.log( window.foo === 'test' );
```

JavaScript's scoping is often a source of confusion. If you are coming from a block-scoped language, this confusion can lead to accidentally global variables, as shown here. Often, this confusion is compounded by imprecise usage of the `var` keyword. For simplicity's sake, the pro JavaScript programmer should always initialize variables with `var`, regardless of scope. This way, your variables will have the scope you expected, and you can avoid accidental globals.

When declaring variables within a function, be aware of the issue of hoisting. Any variable declared within a function has its declaration (not the value it is initialized with) hoisted to the top of the scope. JavaScript does this to ensure that the variable's name is available throughout the scope.

Especially when we combine scope with the concept of context and closures, discussed in the next two sections, JavaScript reveals itself as a powerful scripting language.

Context

Your code will always have some form of context (a scope within which the code is operating). Context can be a powerful tool and is essential for object-oriented code. It is a common feature of other languages, but JavaScript, as is often the case, has a subtly different take on it.

You access context through the variable `this`, which will always refer to the context that the code is running inside. Recall that global objects are actually properties of the `window` object. This means that even in a global context, `this` will still refer to an object. Listing 2-7 shows some simple examples of working with context.

Listing 2-7. Examples of Using Functions Within Context and Then Switching Context to Another Variable

```
function setFoo(fooInput) {
    this.foo = fooInput;
}

var foo = 5;
console.log( 'foo at the window level is set to: ' + foo );

var obj = {
    foo : 10
};

console.log( 'foo inside of obj is set to: ' + obj.foo );

// This will change window-level foo
setFoo( 15 );
console.log( 'foo at the window level is now set to: ' + foo );

// This will change the foo inside the object
obj.setFoo = setFoo;
obj.setFoo( 20 );
console.log( 'foo inside of obj is now set to: ' + obj.foo );
```

In Listing 2-7, our `setFoo` function looks a bit odd. We do not typically use `this` inside a generic utility function. Knowing that we were eventually going to attach `setFoo` to `obj`, we used `this` so we could access the context of `obj`. However, this approach is not strictly necessary. JavaScript has two methods that allow you to run a function in an arbitrary, specified context. Listing 2-8 shows the two methods, `call` and `apply`, that can be used to achieve just that.